QUIC Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 17, 2018

D. Tikhonov
LiteSpeed Technologies
November 13, 2017

QMIN: Header Compression for QUIC
draft-tikhonov-quic-qmin-00

Abstract

This specification defines QMIN, a compression format and protocol for HTTP/2 ([RFC7540]) headers. QMIN is based on HPACK ([RFC7541]). The modifications to HPACK are meant to allow robust compression use in QUIC: That is, no head-of-line blocking and low overhead. QMIN is guided by HPACK design principles. It inherits all of HPACK's data structures and retains binary compatibility with it. While designed with QUIC in mind, QMIN can be used in other contexts.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Google QUIC implementation uses HPACK to compress HTTP headers. HTTP headers for all requests and responses are sent on a dedicated stream. This introduces head-of-line (HoL) blocking: if this stream is blocked due to packet loss, all HTTP messages whose compressed headers follow the lost packet in the stream are stalled. Solving the HoL problem has been one of the goals of the IETF QUIC Working Group.

QMIN solves the HoL problem and has the following beneficial properties:

- The compression logic is mostly contained in the encoder, keeping the decoder simple.
- QMIN is transport-independent.
- Memory penalty over HPACK is manageable (Section 6.4).
- QMIN and HPACK are interoperable (Section 10).

2. Overview

The QMIN innovation is in using a *checkpointed* dynamic table, with the encoder always aware whether the decoder possesses the dynamic table entry (from here on, simply "entry") necessary for decoding a header. The encoder learns this information from messages carried on a single dedicated control stream. The reliable nature of this stream guarantees serialized protocol operation.

In request and response streams, header blocks use either literal representations or references to entries that are known to exist in the decoder table. Dynamic table changes are communicated via the control stream. The process of decoding header blocks does not change the decoder state, thus avoiding the HoL blocking.

QMIN inherits HPACK's data structures and encoding formats (see [RFC7541]).

In addition, *checkpoints* are introduced. A checkpoint is used to track entries added to the dynamic table and streams that reference those entries.

Checkpoints are ordered in a list, from newest to oldest. A new checkpoint gets appended to the "new" end of the checkpoint list.

The encoder always has a checkpoint in the NEW state. Flushing a checkpoint is a two-step operation. First, a FLUSH_CHKPOINT command is sent to the decoder. At that time, the encoder's NEW checkpoint becomes PENDING. The decoder moves its NEW checkpoint directly to LIVE and responds with ACK_FLUSH message. When the encoder receives this message, its PENDING checkpoint becomes LIVE and entries associated with this checkpoint become available for encoding.

The encoder always has exactly one NEW checkpoint, zero or one PENDING checkpoints, and zero or more LIVE and DEAD checkpoints. The decoder has exactly one NEW checkpoint and zero or more LIVE checkpoints.

Unused entries are evicted indirectly, by dropping checkpoints. Before a checkpoint can be dropped, its state is changed to DEAD: the encoder cannot use an entry for encoding that is not referenced by a LIVE checkpoint. Changing a checkpoint's state to DEAD allows the checkpoint to age out. The encoder can decide to drop a DEAD checkpoint when it is no longer referenced by any active streams. See Section 3.

The control stream is used to notify the encoder that the peer is done decoding HTTP headers for a stream using the STREAM_DONE message. The encoder uses this information to track which checkpoints can be dropped.

When a checkpoint is dropped, the table entries it references are checked: if an entry is no longer referenced by any checkpoint, the entry is evicted. The encoder sends the DROP_CHKPOINT command to the decoder when it drops a checkpoint; no acknowledgement for this command is necessary.

Dropping a checkpoint and the entries associated with it is not limited to just the oldest checkpoint; any DEAD checkpoint -- as long as state transition rules are followed -- may be dropped. This flexibility permits the encoder to use a number of strategies for entry eviction.

As long as the maximum dynamic table size is observed, new checkpoints can be created; no upper limit on the number of checkpoints is specified. A well-balanced spread of checkpoints permits the encoder to recycle entries effectively.

The HPACK index address space stays the same. The static table stays as-is. Indices are unique between all checkpoints. An index can be reused once no checkpoint references it.

3. Checkpoint States

A checkpoint can be in one of several states. It goes through these states in order, without skipping any, throughout its lifetime.

On the encoder, the checkpoint states are:

- NEW
- PENDING
- LIVE
- DEAD

On the decoder, only two states are used:

- NEW
- LIVE

3.1. Checkpoint State: NEW

Applicability: encoder and decoder.

All newly reused or inserted entries are referred to by the NEW checkpoint. There is always a NEW checkpoint. Whenever this checkpoint changes state, a new NEW checkpoint is created.

The encoder and the decoder both begin with an empty NEW checkpoint.

3.2. Checkpoint State: PENDING

Applicability: encoder only.

At some point, the encoder may want to flush new entries. It then changes the NEW checkpoint state to PENDING and issues the FLUSH_CHKPOINT command. The entries in the PENDING checkpoint cannot be used for encoding yet; the encoder waits for ACK_FLUSH message. Upon receipt of this message, the PENDING checkpoint changes to the LIVE state.

There can be at most one PENDING checkpoint.

3.3. Checkpoint State: LIVE

Applicability: encoder and decoder.

Entries that were added to the dynamic table when this checkpoint was in the NEW state can now be used to encode and decode headers. The decoder moves its NEW checkpoint to LIVE when it receives the

FLUSH_CHKPOINT command. The encoder moves its PENDING checkpoint to LIVE when it receives the ACK_FLUSH message.

Other than the maximum table size, the number of LIVE checkpoints is not limited.

3.4. Checkpoint State: DEAD

Applicability: encoder only.

To evict old entries, the encoder marks a LIVE checkpoint as DEAD. (An entry that is not referenced by any LIVE checkpoint cannot be used for header encoding. Marking a checkpoint DEAD allows entries to age out.) When all streams whose header blocks were encoded using entries referenced by this checkpoint have been closed, the checkpoint is destroyed and the DROP_CHKPOINT message is sent to the decoder.

There can be any number of DEAD checkpoints.

4. Control Stream

The control stream is used to carry messages to the encoder and the decoder. This is the only way that dynamic table changes are communicated to the decoder.

The messages are either

- Commands issued by the encoder to the decoder;
- Acknowledgements issued by the decoder; or
- Stream processed notifications sent to the encoder.

The format of the messages is similar in structure to the format of the encoded header fields in the header block as specified in HPACK (RFC 7541, Section 6). The same variable-length integer encoding mechanism is used (RFC 7541, Section 5).

4.1. Encoder Commands

The encoder issues the following commands:

4.1.1. INSERT_ENTRY

This message is sent by the encoder at the same time it creates a new indexed entry in its dynamic table. The smallest unused index in the address space ([62 - oO]) MUST be assigned to the new entry.

The decoder creates the new entry in the table, but does not make the entry available for decoding yet. If indexed name representation is used, but the decoder does not have this entry already referenced by its NEW checkpoint, it MUST treat it as an error.

The format of this message is identical to HPACK's Literal Header Field Representation (RFC 7541, Section 6.2).

```
     0   1   2   3   4   5   6   7
   +---+---+---+---+---+---+---+---+
   | 0 | 1 |      Index (6+)       |
   +---+---+-----------------------+
   | H |     Value Length (7+)     |
   +---+---------------------------+
   | Value String (Length octets)  |
   +-------------------------------+
```

Figure: Insert Entry - Indexed Name

```
     0   1   2   3   4   5   6   7
   +---+---+---+---+---+---+---+---+
   | 0 | 1 |           0           |
   +---+---+-----------------------+
   | H |     Name Length (7+)      |
   +---+---------------------------+
   |  Name String (Length octets)  |
   +---+---------------------------+
   | H |     Value Length (7+)     |
   +---+---------------------------+
   | Value String (Length octets)  |
   +-------------------------------+
```

Figure: Insert Entry - New Name

4.1.2. REUSE_ENTRY

This message is issued instead of INSERT_ENTRY whenever the encoder uses an indexed representation from an existing LIVE checkpoint to encode a header and this index has not yet been added to the NEW checkpoint.

Upon receipt of the REUSE_ENTRY command, the decoder creates a reference to the corresponding entry in its NEW checkpoint.

The encoder MUST NOT issue multiple REUSE_ENTRY commands for the same entry in the context of the same NEW checkpoint. If the decoder receives the REUSE_ENTRY message that specifies an index already referenced by its NEW checkpoint, it MUST treat it as an error. If a

non-existent index is specified, the decoder MUST treat is as an error.

The format of this message is identical to HPACK's Indexed Header Field Representation (RFC 7541, Section 6.1).

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 1 |        Index (7+)         |
+---+---------------------------+
```

Figure: Reuse Entry Message

4.1.3. FLUSH_CHKPOINT

When the encoder wants to start using entries associated with the NEW checkpoint, it moves it from NEW to PENDING state and issues the FLUSH_CHKPOINT command.

The decoder moves its checkpoint from NEW to LIVE: all newly inserted entries become available for decoding.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
+---+---------------------------+
```

Figure: Flush Checkpoint

4.1.4. DROP_CHKPOINT

When a DEAD checkpoint is no longer referenced by any streams, the encoder MAY drop it. This means evicting all dynamic table entries whose reference counts have gone to zero and issuing the DROP_CHKPOINT command.

The ID of the checkpoint to drop is its current position in the checkpoint list, from oldest to newest. Thus, the oldest checkpoint has ID 0, second-oldest has ID 1, and so on.

The decoder performs the same operation as the encoder: decrements reference counts of dynamic table entries -- evicting those whose reference counts are now zero -- and drops the specified checkpoint.

```
  0   1   2   3   4   5    6    7
+---+---+---+---+---+---+----+----+
| 0 | 0 | 0 | 0 | 0 | 1 | ID (2+) |
+---+------------------------+----+
```

Figure: Drop Checkpoint

4.2. Decoder Messages

The decoder sends replies to one of the encoder commands.

4.2.1. ACK_FLUSH

The decoder SHOULD inform the encoder that it has performed the flush using ACK_FLUSH message. The encoder's PENDING checkpoint becomes LIVE when this acknowledgement is received.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
+---+---------------------------+
```

Figure: Ack Flush

4.3. Stream Notification Commands

4.3.1. STREAM_DONE

When all HTTP headers for a stream have been decoded, this message is sent to inform the encoder that the peer is done with the stream. This allows the encoder to decrement its reference counts, potentially triggering a checkpoint flush or a checkpoint drop.

It is preferable to send this message as soon as possible. For example, one does not have to wait until stream FIN is read if HTTP headers have been decoded and there are no trailers.

```
  0   1   2   3   4   5    6     7
+---+---+---+---+---+----+-----+-----+
| 0 | 0 | 0 | 0 | 1 | Stream ID (3+) |
+---+--------------------------------+
```

Figure: Stream Done

The client knows that the server is done with the request if the stream is reset or it has read all of the response. A QMIN implementation SHOULD use this knowledge to let the encoder know that

the stream is done. The encoder SHOULD use the earliest indicator to move its mechanisms along. Any subsequent indicators are no-ops.

4.4. Expansion

Two bit patterns are still available to the command coding scheme: 001 and 00000001. The former is used to encode the dynamic table size update by HPACK (RFC 7541, Section 6.3). There is no inherent limitation in QMIN as to why it could not support this command.

5. Header Encoding

The headers are encoded in the same way they are encoded by HPACK, except QMIN does not support the dynamic table size update specified in RFC 7541, Section 6.3 in the headers block. This is because header block decoding is not to change the decoder state.

6. Table Size Calculation

HPACK defines the dynamic table size as "the sum of the size of its entries." (RFC 7541, Section 4.1). QMIN's dynamic table entry carries another element -- reference count -- which increases the entry size.

QMIN introduces checkpoints, whose size should also be accounted for. A decoder-side checkpoint keeps track of the index values created or reused when it was NEW.

6.1. Entry Size

A QMIN entry contains a reference count, which makes it larger that the HPACK entry. Using a standard integer size, the QMIN entry overhead is set to 36 bytes: 32 bytes overhead of the HPACK entry plus four bytes for the additional reference count field. Thus, the QMIN entry size is the sum of the entry name size, the entry value size, and 36.

6.2. Checkpoint Size

QMIN uses the smallest possible available value (Section 4.1.2) in the index address space for new entries. Therefore, the total number of index values is at most the value of the largest index in use. A checkpoint can track indices via a bitmask: 1 bit per index. The size of a checkpoint, then, is defined as

```
    (Highest Index Value - 62) / 8 + 128
```

The additional 128 bytes is the checkpoint overhead.

6.3. Overall Table Size

The decoder table size is calculated as number of entries times the entry size as calculated in Section 6.1 plus the number of checkpoints times the checkpoint size as calculated in Section 6.2.

6.4. Comparison with HPACK

In HPACK, a table with 700 dynamic entries and 35,000 bytes allocated to header names and values is

```
    700 * 32 + 35,000 = 57,400 bytes.
```

The same table in QMIN with 10 checkpoints is

```
    700 * 36 + 35,000 + 10 * ((1000 / 8) + 128) = 62,730 bytes.
```

This is a 9% increase in memory consumption.

7. Encoding Process

Given a header field to compress, the encoder returns the compressed representation of it. In addition, it may emit one or more commands that should be sent on the control stream.

7.1. Indexable Header Fields

An indexable header field is that which the user specifies as "with indexing" (RFC 7541, Section 6.2).

7.1.1. New Index

If no matching entry is found, a new entry is created, its ID is recorded in the NEW checkpoint, and the encoder emits the INSERT_ENTRY command.

If the encoded name component refers to an existing entry, this entry is reused as described in Section 7.1.2.

7.1.2. Existing Index

An indexable header field causes the encoder to search the table. If an existing dynamic table entry is found that is referenced by at least one LIVE checkpoint, it can be used to encode the header field. The encoder records a reference to the stream using this entry in one of the checkpoints. (Which checkpoint to select can be decided based on strategy. See Section 9.1).

If the NEW checkpoint does not have a reference to this entry, the reference is recorded in the NEW checkpoint and the REUSE_ENTRY command is emitted.

7.2. Non-indexable Header Fields

Non-indexable header fields are compressed the same way as HPACK (RFC 7541, Sections 6.2.2 and 6.2.3). The encoder state is not changed. No command is emitted.

7.3. When Maximum Table Size Is Reached

When the encoder table reaches its maximum size, further insertions into the dynamic table are not possible. In this case, the encoder compresses header fields without inserting or reusing entries and without emitting any commands.

A simple recovery strategy is to mark one or more checkpoints DEAD immediately.

Alternatively, the existing table may provide an acceptable compression level. It may be more efficient to wait until this level falls below a threshold before marking checkpoints DEAD, as it may become possible to drop an already-DEAD checkpoint before the threshold is reached.

The encoder SHOULD try to avoid reaching a point when it can no longer insert new entries. See Section 9.

7.4. Memory Cost of Flushing

Because flushing automatically creates a new NEW checkpoint, it is possible to get into a situation where a flush is not possible due to the memory constraint. If inserting a new entry would result in subsequent inability to flush, the encoder SHOULD flush instead.

8. Decoding Process

All header field representations defined in HPACK (Section 6 of [RFC7541]) are used as-is. Dynamic size update (Ibid., Section 6.3) or an unknown command MUST be treated as an error.

The decoder looks up dynamic entries in its table when it is given a header list to decode. If corresponding entry is not found or if it is found but not referred to by any of LIVE checkpoints, this MUST be treated as an error.

9. Encoder Strategies

9.1. Flushing and Dropping

The encoder decides when to flush checkpoints and when to declare them dead. Flushing SHOULD occur when enough new entries have been created to try to reuse them. Marking checkpoints as DEAD SHOULD happen before the table size is exhausted.

If an entry used to encode a header field is referenced to by more than one LIVE checkpoint, one of them is selected to refer to the stream ID whose header field has been encoded. Which LIVE checkpoint to pick is a decision that also affects compression performance.

Several strategies are outlined below.

9.1.1. Simple Strategy

The encoder picks a number of streams to use as a threshold for flushing checkpoints. Every time header blocks for N streams have been encoded, flush.

The encoder picks the oldest checkpoint to mark as DEAD. It does so when table size reaches some proportion, let's say 3/4, of the maximum table size.

The newest LIVE checkpoint that references an entry used for encoding is picked to record the stream ID.

This strategy is estimated to work well most of the time due to the temporal aspect of the checkpoint dropping policy. When a connection is used to serve a small number of requests, however, the compression will be overall suboptimal, as the initial period when no dynamic table is available for encoding is amortized poorly.

9.1.2. Rule-Based Strategy

Heuristic rules may provide performance improvement over the simple strategy above. For example:

- Flush very often, perhaps once for every new stream, when the number of dynamic entries is very small (such as when the encoder has just been instantiated). Since the table size is likely to be small when only few dynamic entries exist, one can fit a lot of checkpoints and still be able to add new entries to the table. These early-flushed checkpoints will also be easier to drop later, as they are not referenced by many streams.

- Flush when the number of newly added entries is 1/10 of the number of existing entries. When this many new entries have been added, it is a likely indicator making them available for encoding will improve overall compression.
- When declaring a checkpoint DEAD:

  * Pick a LIVE checkpoint that is referenced by the fewest existing streams; or
  * Pick a LIVE checkpoint that references the largest number of old entries, where an "old" entry is that which has not been used for encoding in a period of some number of checkpoints.

Other rules are possible.

9.1.3. Feedback-Based Strategy

The goal of QMIN is to produce the best compression. The compression level can be computed by dividing the sum of the sizes of all header fields submitted for compression by the number of compressed bytes returned *plus* the size of all commands sent to the decoder. A checkpoint can be taken as unit of time and a decaying average can be computed.

Availability of entries that can be used for compression directly affects compression performance. This availability, in turn, is a function of how often checkpoints are flushed and which checkpoints are marked for deletion. Flushing very often costs memory; infrequent flushing delays entry availability.

It is possible to come up with a dynamic function that adjusts these parameters based on feedback: the compression performance.

9.2. Control Channel Cost

Sending commands on the control channel affects the overall compression level. Sending an INSERT_ENTRY command for a header field that is never reused is more expensive than not inserting the field at all. A single large, ever-changing HTTP header (for example, session state in a cookie) could defeat the compression mechanism. The encoder SHOULD prevent this from happening.

Since a header field that repeats is likely to repeat more than once, a simple conservative approach is never to insert a header field that is not known to have repeated. Because HTTP header names are relatively small and not as numerous as the header values, it is possible to maintain a history of a number of recently compressed header fields. (To use less memory, hashes of header values, instead of the values themselves, can be stored.) The encoder can consult

this history and only issue an INSERT_ENTRY command if the header field has been seen before.

10. HPACK Interoperability

Because QMIN uses the same binary format as HPACK, the two are interoperable. This makes it possible for peers to use the current HTTP/QUIC HPACK mechanism to talk to peers that use QMIN. It is useful: all implementations do not have to start using QMIN at the same time.

For this to work, four things must be true:

1. The HPACK side must advertise maximum dynamic table size of zero.
2. The HPACK side must not send dynamic table size updates.
3. The HPACK side must consume and discard data sent on the control stream. This is so that QMIN sender does not get stuck when it reaches the stream flow control limit.
4. The HPACK side must assume that peer's dynamic table size is zero. This is to prevent HPACK encoder from relying on dynamic entries.

(1) and (2) are already true according to [I-D.ietf-quic-http]. (3) and (4) are trivial modifications.

11. Implementation Notes

11.1. Control Messages Made Easy

Since INSERT_ENTRY and REUSE_ENTRY messages are identical to the encoded header field representation, the latter can be placed onto the control stream verbatim. Generate once, use twice.

12. QMIN Drawbacks

The following QMIN properties affect compression negatively:

- All insertion commands are duplicated: they are sent both as literal representation in headers block and as insertion commands on the control stream.
- A new entry cannot be used until the checkpoint is flushed and the encoder receives ACK_FLUSH message. Until that time, the header field literal representation must be used for subsequent encodings.

13. Acknowledgements

QMIN is based on HPACK ([RFC7540]); I am thankful to its authors.

Observations of the following members of the IETF QUIC WG have been particularly insightful:

- Alan Frindell;
- Charles 'Buck' Krasic; and
- Mike Bishop.

Finally, my colleagues at LiteSpeed Technologies reviewed the rough draft and provided valuable feedback:

- George Wang;
- Ron Saad.

14. References

14.1. Normative References

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

14.2. Informative References

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-07 (work in progress), October 2017.

Author's Address

Dmitri Tikhonov
LiteSpeed Technologies

Email: dtikhonov@litespeedtech.com